

Word equations in linear space*

Artur Jeż

Institute of Computer Science, University of Wrocław, Poland

February 3, 2017

Abstract

Word equations (in free semigroup) are an important problem on the intersection of formal languages and algebra. Given two sequences consisting of letters and variables we are to decide whether there is a substitution for the variables that turns this formal equation into true equality of strings. The exact computational complexity of this problem remains unknown, with the best known lower and upper bounds being NP and PSPACE , respectively. Recently, a novel technique of recompression was applied to this problem, simplifying the known proofs and lowering the space complexity to (nondeterministic) $\mathcal{O}(n \log n)$, where n is the length of the input equation. In this paper we show that word equations are in nondeterministic linear space, thus the language of satisfiable word equations is context-sensitive. We use the known recompression-based algorithm and additionally employ Huffman coding for letters. The proof, however, uses analysis of how the fragments of the equation depend on each other as well as a new strategy for nondeterministic choices of the algorithm, which uses several new ideas to limit the space occupied by the letters.

1 Introduction

Solving *word equation* was an intriguing problem since the beginning of computer science, investigated initially due to its ties to Hilbert's 10th problem. Initially it was conjectured that this problem is undecidable, which was disproved by Makanin [10]. At the beginning little attention was given to computational complexity of Makanin's algorithm and the problem itself; these questions were reinvestigated in the '90 [6, 18, 9], culminating in the EXPSPACE implementation of Makanin's algorithm by Gutiérrez [5].

The connection between compression and word equations was first observed by Plandowski and Rytter [16], who showed that a length-minimal solution of size N has a compressed representation of size $\text{poly}(n, \log N)$. Plandowski further explored this approach [14] and finally proposed a PSPACE algorithm [13]. This is the best bound up to date, though a simpler PSPACE solution with smaller space consumption based on compression was proposed by Jeż [8].

All mentioned algorithms extend (in perhaps nontrivial ways) to various important scenarios: groups [11, 2, 4], representation of all solutions [15, 8, 17], traces [12, 3], graph groups [1] equations over terms [7] and many others.

On the other hand, word equations are only known to be NP-hard , which is easy to see, as they generalise integer programming; and it was conjectured by Plandowski that this problem is in fact NP-complete .

While the computational complexity of word equations remains unknown, its exact space complexity is intriguing as well: Plandowski [13] gave no explicit bound on the space usage of his algorithm, a rough estimation is $\mathcal{O}(n^5)$ and the recent solution of Jeż [8] gives a (nondeterministic) $\mathcal{O}(n \log n)$ space complexity. Moreover, for $\mathcal{O}(1)$ variables a linear bound on space complexity was shown [8]; recall that languages recognisable in nondeterministic linear space coincide with those generated by context-sensitive grammars.

*This work was supported under National Science Centre, Poland project number 2014/15/B/ST6/00615.

In this paper we show that satisfiability of word equations can be tested in nondeterministic linear space in terms of number of bits of the input, thus showing that the language of satisfiable word equations is context-sensitive (and by the famous Immerman–Szelepcsényi theorem, also the language of unsatisfiable word equations is context sensitive). The employed algorithm is a (variant of) algorithm of Jež [8], which additionally uses Huffman coding for letters in the equation. On the other hand, the actual proof uses different encoding of letters, which extends the ideas used in a (much simpler) proof in case of $\mathcal{O}(1)$ variables [8, Section 5]; the main new ingredient is a different strategy of compression: roughly speaking, previously a strategy that minimised the length of the equation was used. Here, a much more sublime strategy is used, it simultaneously minimises the size of a particular bit encoding, enforces that changes in the equation (during the algorithm) are local, and limits the amount of new letters that are introduced to the equation.

The algorithm does not assume that each letter and variable in the input is encoded using the same number of bits, and an arbitrary encoding can be used, so in particular, the Huffman coding (so the most efficient one) is allowed.

2 The (known) algorithm

We now present (a slight variant of) the algorithm of Jež [8] and the notions necessary to understand, how it works. The proofs are omitted, yet they should be intuitively clear.

2.1 Notions

The word equation is a pair (U, V) , often written as $U = V$, where $U, V \in (\Gamma \cup \mathcal{X})^*$ and Γ and \mathcal{X} are disjoint alphabets of *letters* and *variables*; variables and letters are collectively called *symbols*; we use Γ instead of standard Σ as we heavily employ the summation notation. By n_X we denote the number of occurrences of X in the (current) equation; the algorithm guarantees that n_X does not change till X is removed from the equation, in which case n_X is set to 0. A *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \rightarrow \Gamma'^*$, where $\Gamma' \supseteq \Gamma$ and $S(a) = a$ for every $a \in \Gamma$, a substitution naturally extends to $(\mathcal{X} \cup \Gamma)^*$. A *solution* of an equation $U = V$ is a substitution S such that $S(U) = S(V)$. We allow the solution to use letters that are not present in the equation, this does not change the satisfiability of the equation, (as all such letters can be changed to a fixed letter from Γ , and the obtained substitution is still a solution). However, the construction and proofs become easier and more transparent, when we allow the usage of such letters. In particular, the exact set Γ' is usually given implicitly: as the set of letters used by the substitution. We sometime use solutions S that are *length-minimal*, i.e. for every solution S' it holds that $|S(U)| \leq |S'(U)|$.

For succinctness, we use the notion *substring* to denote a sequence of letters, also ones occurring in the equation. A *factor* is a sequence of letters and variables occurring in the equation. A *block* is a sequence a^ℓ for $\ell \geq 1$ that cannot be extended to the left nor to the right with a .

As we deal with linear-space, the actual encoding used by the input equation is important. We assume only that the input is given by a fixed (uniquely decodable) coding: each symbol in the input is always given by the same bitstring and given the bitstrings representing the sides of the equation there is only one pair of strings (over $\Gamma \cup \mathcal{X}$) that is bit-coded in this way. It is already folklore that among such codes the Huffman code yields the smallest space consumption (counted in bits) and moreover the Huffman coding can be efficiently computed, also in linear space. As we focus on space counted in bits and use encodings, by $||\alpha||$ we denote the space consumption of the encoding of α , the encoding shall be always clear from the context. Furthermore, whenever we talk about space complexity, it is counted in bits.

2.2 Nondeterministic Linear Space

Let us recall some basic facts about the nondeterministic space-bounded computation. A non-deterministic procedure is *sound*, when given a unsatisfiable word equation $U = V$ it cannot transform it to a satisfiable one, regardless of the nondeterministic choices; such a procedure is

complete, if given a satisfiable equation $U = V$ for some nondeterministic choices it returns a satisfiable equation $U' = V'$. A composition of sound (complete) procedures is sound (complete, respectively).

As we aim at linear-bounded computation, it is enough to show such a bound for one particular computation: when the bound is known, we can limit the space available to the algorithm and simply reject the computation that exceeds the allowed space. Thus we imagine the computation of our algorithm as if it had extra knowledge, that allows it to make the nondeterministic choices appropriately and we bound the space computation only in the case of those appropriate nondeterministic choices. In particular, the subprocedures described in the next section are written ‘as if’ the algorithm knew a particular solution of the current equation.

2.3 The algorithm

We use (a slight variant of) recompression algorithm [8]. It is based on the following two operations, which are conceptually applied on $S(U)$ and $S(V)$: given a string w and alphabet Γ

- the Γ block compression of w is a string w' obtained by replacing every block a^ℓ , where $a \in \Gamma$ and $\ell \geq 2$, with a (fresh) letter a_ℓ ;
- the (Γ_ℓ, Γ_r) pair compression of w , where Γ_ℓ, Γ_r is a partition of Γ , is a string w' obtained by replacing every occurrence of a pair $ab \in \Gamma_\ell \Gamma_r$ with a fresh letter c_{ab} .

A ‘fresh’ letter means that it is not currently used in the equation, nor in Γ , yet each occurrence of a fixed ab is replaced with the same letter. The a_ℓ and c_{ab} are just notation conventions, the actual letters in w' do not store the information, from what they were obtained. For shortness, we call Γ block compression the Γ -compression or block compression, when Γ is clear from the context; similar convention applies to (Γ_ℓ, Γ_r) pair compression. We say that a pair $ab \in \Gamma_\ell \Gamma_r$ is *covered* by (Γ_ℓ, Γ_r) -compression.

The intuition is that the algorithm aims at performing those compression operations on $S(U)$ and $S(V)$ and to this end it modifies the equation a bit and then performs the compression operations on U and V (and conceptually also on the solution, i.e. on $S(X)$ for each variable X). Below we describe, how it is performed on the equation.

Block compression For the equation $U = V$ and the alphabet Γ of letters in this equation for each variable X we first guess the first and last letter of $S(X)$ as well as the lengths ℓ, r of the longest prefix consisting only of a , called a -prefix, and b -suffix (defined similarly) of $S(X)$. Then we replace X with $a^\ell X b^r$ (or $a^\ell b^r$ or a^ℓ when $S(X) = a^\ell X b^r$ or $S(X) = a^\ell$); this operation is called *popping a -prefix and b -suffix*. Then we perform the Γ -block compression on the equation (this is well defined, as we can treat variables as symbols from outside Γ).

Algorithm 1 BlockComp(Γ)

Require: Γ is the set of letters in $U = V$

- 1: **for** $X \in \mathcal{X}$ **do**
 - 2: let a, b be the first and last letter of $S(X)$
 - 3: guess $\ell \geq 1, r \geq 0$ $\triangleright S(X) = a^\ell w b^r$, where w does not begin with a nor end with b
 - 4: replace each X in U and V by $a^\ell X b^r$ $\triangleright S(X) = a^\ell w b^r$ changes to $S(X) = w$
 - 5: **if** $S(X) = \epsilon$ **then** remove X from U and V \triangleright Guess
 - 6: **for** each letter $a \in \Gamma$ and each $\ell \geq 2$ **do**
 - 7: replace every block a^ℓ occurring in U or V by a fresh letter a_ℓ
-

Pair compression For the alphabet Γ , which will always be the alphabet of letters in the equation right before the block compression we partition Γ into Γ_ℓ and Γ_r (in a way described in detail in Section 3.2, but a random partition will do) and then for each variable X guess whether $S(X)$ begins with a letter $b \in \Gamma_r$ and if so, replace X with bX or b , when $S(X) = b$, and then do

a symmetric action for the last letter and Γ_ℓ ; this operation is later referred to as *popping letters*. Then we perform the (Γ_ℓ, Γ_r) compression on the equation.

Algorithm 2 PairComp(Γ_ℓ, Γ_r)

Require: Γ_ℓ, Γ_r are disjoint

```

1: for  $X \in \mathcal{X}$  do
2:   let  $b$  be the first letter of  $S(X)$  ▷ Guess
3:   if  $b \in \Gamma_r$  then
4:     replace each  $X$  in  $U$  and  $V$  by  $bX$  ▷ Implicitly change  $S(X) = bw$  to  $S(X) = w$ 
5:     if  $S(X) = \epsilon$  then remove  $X$  from  $U$  and  $V$  ▷ Guess
6:   let  $a$  be the ... ▷ Perform a symmetric action for the last letter
7:   for  $ab \in \Gamma_\ell \Gamma_r$  do
8:     replace each explicit  $ab$  in  $U$  and  $V$  by a fresh letter  $c$ 

```

The algorithm LinWordEqSat works in *phases*, until a trivial equation (i.e. with both sides of length 1) is obtained: in a single phase it establishes the alphabet Γ of letters in the equation, performs the Γ compression and then repeats: guess the partition of Γ to Γ_ℓ and Γ_r (see Section 3.2) and perform the (Γ_ℓ, Γ_r) -compression, until each pair $ab \in \Gamma^2$ was covered by some partition.

Algorithm 3 LinWordEqSat

```

1: while  $|U| > 1$  or  $|V| > 1$  do
2:    $\Gamma \leftarrow$  letters in  $U = V$ 
3:   BlockComp( $\Gamma$ )
4:   while some pair in  $\Gamma^2$  was not covered do
5:     partition  $\Gamma$  to  $\Gamma_\ell$  and  $\Gamma_r$  ▷ Guess
6:     PairComp( $\Gamma_\ell, \Gamma_r$ )

```

2.4 Correctness

Given a solution S we say that nondeterministic choices *correspond* to S , if they are done as if LinWordEqSat knew S . For instance, it guesses correctly the first letter of $S(X)$ or whether $S(U) = \epsilon$. (Note that the choice of a partition does not fall under this category.)

All of our procedures are sound and complete, furthermore they transform the solutions in the sense described in the below Lemma 1.

Lemma 1 ([8, Lemma 2.8 and Lemma 2.10]). *BlockComp(Γ) is sound and complete; to be more precise, for any solution S of an equation $U = V$ for the nondeterministic choices corresponding to S the returned equation $U' = V'$ has a solution S' such that $S'(U')$ is the Γ -compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the a -prefix and b -suffix, where a is the first letter of $S(X)$ and b the last, and then performing the Γ -compression.*

When Γ_ℓ and Γ_r are disjoint, the PairComp(Γ_ℓ, Γ_r) is sound and complete; to be more precise, for any solution S of an equation $U = V$ for the nondeterministic choices corresponding to S the returned equation $U' = V'$ has a solution S' such that $S'(U')$ is the (Γ_ℓ, Γ_r) -compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the first letter of $S(X)$, if it is in Γ_r , and the last, if it is in Γ_ℓ , and then performing the (Γ_ℓ, Γ_r) -compression.

The solution S' from Lemma 1 is called a *solution corresponding to S* after (Γ_ℓ, Γ_r) -compression (Γ -compression, respectively); we also talk about a *solution corresponding to S* , when the compression operation is clear from the context and also extend this notion to a solution corresponding to S after a phase. What is important later on is how S' is obtained from S : it is modified as if the subprocedures knew first/last letter of $S(X)$ and popped appropriate letters from the variables and then compressed pairs/blocks in substitution for variables.

Lemma 1 shows soundness of the whole procedure, for the completeness we will need a simple observation that iterating the compression operations shortens the string by a constant fraction, thus the length of the length-minimal solution will shorten by a constant fraction after each phase.

Lemma 2. *Let w be a string over an alphabet Γ and w' a string obtained from w by a Γ -compression followed by a sequence of (Γ_ℓ, Γ_r) -compressions (where Γ_ℓ, Γ_r is a partition of Γ) such that each pair $ab \in \Gamma^2$ is covered by some partition. Then $|w'| \leq \frac{2|w|+1}{3}$.*

Proof. Consider two consecutive letters a, b in w . At least one of those letters is compressed during the procedure:

- if $a = b$: In this case they are compressed during the Γ -compression.
- $a \neq b$: At some point the pair ab is covered by some (Γ_ℓ, Γ_r) compression. If any of letters a, b was already compressed then we are done. Otherwise, this occurrence of ab is now compressed.

Hence each uncompressed letter in w (except perhaps the last letter) can be associated with the two letters to the right that are compressed. This means that (in a phase) at least $\frac{2}{3}(|w| - 1)$ letters are compressed and so $|w'| \leq |w| - \frac{1}{3}(|w| - 1)$, as claimed. \square

Theorem 1. *LinWordEqSat is sound, complete and terminates (with appropriate nondeterministic choices) for satisfiable equations. It runs in bitspace that is linear in the size of the input equation.*

The proof is given in Section 3.3.

In the following, we will also need one more technical property of block compression.

Lemma 3. *Consider a solution S during a phase with non-deterministic choices corresponding to S and any corresponding solution S' of $U' = V'$ after the block compression. Then $S'(U')$ has no two consecutive letters $aa \in \Gamma$.*

Proof. This is true immediately after block compression and afterwards no letters from Γ are introduced, only removed. \square

2.5 Compressing blocks in small space

The storage, even in a concise way, of lengths of popped prefixes and suffixes in Γ -compression makes attaining the linear space difficult. This was already observed in [8] and a linear-space implementation of **BlockComp** was provided there. Note that it performs different set of operations, yet the effect is the same as for **BlockComp**. The idea is that instead of explicitly naming the lengths of blocks, we treat them as integer parameters; then we declare, which maximal blocks are of the same length (those lengths depend linearly on the parameters); verifying, whether such a guess is valid is done by writing a system of (linear) Diophantine equations that formalise those equalities and checking, whether it is satisfiable. This procedure is described in detail in [8, Section 4]. In the end, it can be implemented in linear bitspace.

Lemma 4 ([8, Lemma 4.7]). *BlockComp can be implemented in space linear in the bit-size of the given word equation*

2.6 Huffman coding

At each step of the algorithm we encode letters (though not variables) in the equation using Huffman coding. This may mean that when going from $U = V$ to $U' = V'$ the encoding of letters may change and in fact using the former encoding in the latter equation may lead to super-linear space (imagine that we pop from each variable a letter that has a very long code). Thus we comment how to make a transition from $U = V$ to $U' = V'$ in bit-space $\mathcal{O}(\|U = V\| + \|U' = V'\|)$.

Lemma 5. *Given two strings (encoded using some uniquely decodable code), their Huffman coding can be computed in linear bitspace.*

Any subprocedure of LinWordEqSat that transform an equation $U = V$ to $U' = V'$ this subprocedure can be implemented in bit-space $\mathcal{O}(\|U = V\|_1 + \|U' = V'\|_2)$, where $\|\cdot\|_1$ and $\|\cdot\|_2$ are the Huffman codings for letters in $U = V$ and $U' = V'$, respectively.

Proof. A standard implementation of the Huffman coding firstly calculates for each symbol in the text the number of its occurrences, this can be done in linear space, as a symbol plus number of its occurrences takes space linear in the space taken by all those occurrences. Then it iteratively builds an edge-labelled tree with leaves corresponding to original letters. The labels on the path from the root to a leaf a give a code for a . The algorithm takes two letters with the smallest number of occurrences, creates a new node (which is treated further on as a leaf), attaches the two nodes to the new node and labels the edges with 0 and 1. This is iterated till one node is obtained. The tree uses linear space in total and otherwise the used space only decreases. It is easy to see that the whole computation can be done in linear space.

For popping letters in (Γ_ℓ, Γ_r) -compression we use new symbols $X\#0$, $X\#1$ that are bit-encoded as X plus $\mathcal{O}(1)$ bits for letters popped to the left and right and then simply list the letters that are equal. In this way we can compute the Huffman coding after popping and translate to the new encoding. For compression itself, when ab is compressed, we encode it as (ab) , where ‘(’ and ‘)’ are new symbols and then recompute the Huffman coding.

For block compression, Lemma 4 already states that it can be performed in space linear in the bitsize of the old equation, the Huffman coding of the new one can be then computed. \square

3 Space consumption

In order to bound the space consumption, we will use bit-encoding of letters that depends on the current equation. We use the term ‘encoding’ even though it may assign different codes to different occurrences of the same letter, but two different letters never have the same code. Since we are interested in linear space only, we do not care about the multiplicative $\mathcal{O}(1)$ factors in the space consumption and can assume that our code is prefix-free, say by ending each encoding with a special symbol $\$$. We show that such an encoding uses linear space, which also shows that the Huffman encoding of the letters in the equation uses linear space, as Huffman encoding uses the smallest space among the prefix codes.

The idea of our ‘encoding’ is fairly simple: for each letter we establish the part D of the original equation on which it ‘depends’ (this has to be formalised) and encode this letter as $D\#i$, when it is i th in the sequence of letters assigned D ; we prove that letters given the same encoding are indeed the same and formalise the ‘dependency’. The idea of dependency is formalised in Section 3.1, while Section 3.2 first gives the high-level intuition and then the actual estimation of the used space.

For technical reasons we insert into the equation ending markers at the beginning and end of U and V , i.e. write them as $@U@$, $@V@$ for some special symbol $@$. Those markers are going to be completely ignored by the algorithm, yet they are needed for the encoding.

We use a partial order \leq on substrings and factors of a fixed sequence w : $w[i..j] \leq w[i'..j']$ if $i \leq i'$ and $j \leq j'$; when strings are said to be *smaller* or *greater*, this is with respect to this partial order.

3.1 Dependency factors

We associate with each symbol in the equation (including the ending markers) its *dependency factor*, called *depfactor* for short; initially a depfactor of a symbol α is α . In general, a depfactor is a factor of the original equation and it indicates that during the run of LinWordEqSat this letter is uniquely determined by this factor (and the nondeterministic choices of the algorithm), in particular it was obtained as a compression of letters in this factor or popped from this factor. Furthermore, we ensure that when we look at maximal factors of symbols with the same depfactor

then they are the same. To get some rough intuition: when a letter is popped from a variable X , then it depends solely on X and it has a depfactor X . When we compress w to a single letter, the resulting depfactor is a union of all depfactors of letters in w .

Formally, the depfactors are defined as follows: For each occurrence of a symbol α in the initial equation we define its *basic depfactor* α . Basic depfactors are bit-encoded exactly the same as their corresponding symbols; by $|| \cdot ||$ we will denote the bit length of this encoding and call it *weight*. For different occurrences of α in the original equation, the basic depfactors are denoted by the same letters, yet they are different depfactors, which are just represented by the same symbol; this is the same as for different occurrences of letters, which are still denoted by the same letter. A *depfactor*, usually denoted by letters D, D' , etc., is a sequence string of consecutive basic depfactors (from one side of the equation) and is represented as a concatenation of those basic depfactors, depfactors represented by the same factors of the input equation are *similar*, denoted by $D \sim D'$. Every symbol (letter) has exactly one depfactor, say D , we call it a D -symbol (D -letter, respectively); if the depfactor D' can be obtained by concatenating some basic depfactors to a depfactor D (which is denoted by $D' \subseteq D$) then a D' letter is also a sup- D letter the set of those letters is denoted as $\text{sup } D$; by $|D|, |\text{sup } D|$ we denote the number of D -symbols and $\text{sup } D$ -symbols. While D -letters are formally defined in the equation, we use those notions also for the corresponding letters in $S(U)$ and $S(V)$ (the letters that come from variables do not have depfactors). We use a partial order on depfactors: $D \leq D'$ as depfactors if $D \leq D'$ as factors in the input equation; for simplicity, we fix the left and right-hand side of the equation and the factors of the left-hand side are smaller than the ones of the right-hand side.

Given a depfactor D the D -letters shall form a substring in the equation, furthermore, for two depfactors $D \sim D'$ their strings of letters shall be the same. Furthermore, any two depfactors D, D' that have letters in the equation shall be comparable (in the partial order on the depfactors \leq); formally:

- (D1) Given a depfactor D , the D -symbols and $\text{sup } D$ -symbols are factors of the equation.
- (D2) Given two depfactors D, D' that have symbols in the equation, either $D \leq D'$ or $D \geq D'$.
- (D3) For depfactors $D \sim D'$ the strings of D - and D' -letters are the same.

Thanks to (D1) we can enumerate the D letters in a given equation (from left-to-right) and talk about the first, second, etc. D -letter; we denote (and encode) them as $D\#1, D\#2$, etc. and by (D3) those encodings are the same for similar depfactors.

During the algorithm, the depfactors can be summed: a *sum* of depfactors $D \leq D'$, denoted by DD' , is obtained by taking their respecting strings, removing from D the basic depfactors that occur in D' and concatenating the two resulting strings. This operation naturally extends to any number of depfactors $D_1 \leq \dots \leq D_n$ and it is associative.

3.1.1 Assigning depfactors to letters

When an occurrence of X pops a letter this letter gets the (basic) depfactor of this occurrence of X . Whenever we perform the (Γ_ℓ, Γ_r) -compression, then in parallel for each occurrence of a letter $a \in \Gamma_\ell$ with a depfactor D we establish a depfactor D' of symbol to its right (note that it may be a variable or an endmarker) and make DD' the depfactor of a . Then we perform a symmetric action for all occurrences of letters in Γ_r (so D' is to the left now). A simple argument, see Lemma 6, shows that the order of operation (Γ_ℓ or Γ_r first) does not matter.

For Γ compression, we perform in parallel the following operation for each maximal block (perhaps of length 1) of a letter in Γ : given a maximal block a^ℓ with depfactors D_1, \dots, D_ℓ and depfactors D to the left and D' to the right we replace the depfactors of all letters in this a^ℓ by $DD_1 \dots D_\ell D'$.

Lemma 6. *The depfactors assigned before pair compression are the same, regardless of whether the Γ_ℓ or Γ_r letters are considered first.*

Proof. It is enough to show that for three consecutive letters abc the depfactor of b is going to be the same, regardless of whether we consider Γ_ℓ or Γ_r first. If $b \notin \Gamma_\ell \cup \Gamma_r$ then there is nothing to prove; the case $b \in \Gamma_\ell$ and $b \in \Gamma_r$ are symmetric (note that it is always true that $\Gamma_\ell \cap \Gamma_r = \emptyset$), so we consider only the former.

Let the depfactors of b, c be D, D' . If we consider Γ_ℓ first, then in the first step b gets the depfactor DD' and in the second step nothing changes. If we consider first Γ_r and $c \notin \Gamma_r$ then after the first step the depfactors of b, c are still D, D' and in the second step b gets depfactor DD' . If $c \in \Gamma_r$ then in the first step it gets the depfactor DD' and b still has depfactor D . Then in the second step b gets depfactor $D(DD')$, which is the same as DD' , as we remove the duplicates. \square

In the following we will mostly focus on sup- D symbols for basic depfactors D . As they are intervals, we visualize that sup D extends to the neighbouring letters. Thus we will refer to operations of changing the depfactors before the block compression and pair compression as *extending of* depfactor D to new letters; those letters get their depfactors extended. Note that this notion does not apply to the case when we pop letters from variables. Note that the same operation may extend D and D' to the same letter.

The crucial task is to show that the way of assigning and changing the depfactors is well-defined, that is, that the resulting depfactors satisfy the conditions (D1–D3).

Lemma 7. *(D1–D3) holds during LinWordEqSat.*

Proof. Concerning (D1), consider first this claim for sup- D -symbols, where D is a basic depfactor, we show it by induction; this is true at the beginning. If the depfactors are summed, a letter adjacent to a sup- D -symbol can become a sup- D -symbol (this can be iterated when the depfactors are changed before the blocks compression), which is fine. During the compression, we compress symbols of the same depfactor, so this is fine. When we pop a letter, it is of the same depfactor as its variable, this variable is a sup- D -symbol and by inductive assumption it was part of the factor of sup- D -symbols, which shows the claim.

We now show by induction that for basic depfactors $D \leq D'$ it holds that $\text{sup } D \leq \text{sup } D'$. Clearly this holds at the beginning, as there is only one sup- D and sup- D' -symbol. Consider the moment, in which the condition $\text{sup } D \leq \text{sup } D'$ is first violated, by symmetry it is enough to consider the case in which the first sup- D' -symbol is smaller than first sup- D -symbol. If this letter was just popped then it cannot be popped to the right, as its variable is a sup- D' symbol as well. So it was popped to the left. But then the variable that popped it was a D' -symbol and by induction assumption it was greater than $\text{sup } D$, so it had a sup D -symbol to its left, contradiction. The other option is that this happened when a context of a letter a was changed so that a became a sup- D' letter. But then the symbol to a 's right was a sup D' -symbol and by induction assumption either this letter was a sup- D letter or some letter to the left of it was; in both cases the letter also became a sup D -letter.

We are now ready to show (D1) for sup- D -symbols for an arbitrary depfactor D . Let $D_1 D_2 \cdots D_m$, be the consecutive basic depfactors of D 's side of the equation and let $D = D_k \cdots D_\ell$ for $1 \leq k \leq \ell \leq m$. A symbol is in $\text{sup } D$ if it is in each $\text{sup } D_i$ for $i = k, \dots, \ell$ and in none $\text{sup } D_i$ for $i = 1, \dots, k-1, \ell+1, \dots, m$. Observe that as $D_1 \leq D_2 \leq \cdots \leq D_m$, if a symbol is in $\text{sup } D_k$ and in some $\text{sup } D_i$ for $i < k$ then it is also in D_{k-1} , similarly, if a symbol is in $\text{sup } D_\ell$ and $\text{sup } D_i$ for $i > \ell$ then it is in $\text{sup } D_{\ell+1}$. Thus $\text{sup } D$ is obtained as $D' = \bigcap_{i=k}^{\ell} \text{sup } D_i$ (which is an intersection of intervals, so an interval) minus $\text{sup } D_{k-1}$ and $\text{sup } D_{\ell+1}$, which corresponds to a deletion of a prefix and a suffix of D' , so the result is still an interval.

In the following, it is useful to define the sub- D letters, which are a dual notion to sup- D letters: if a $D' \supseteq D$ then a D -letter is a sub- D' letter, the set of such letters is denoted as $\text{sub } D'$.

We show a strong auxiliary claim:

Auxiliary Claim. *Given two similar depfactors $D \sim D'$ the sub D and sub D' are factors of the equation, the corresponding symbols in them are the same and have similar depfactors.*

Note that Auxiliary Claim is stronger than (D3), so in particular it implies it.

We prove Auxiliary Claim by induction: As $D \sim D'$ they are represented by the same factors and $\text{sub } D$ and $\text{sub } D'$ are those factors at the beginning, so the claim trivially holds at the beginning. If any variable in $\text{sub } D$ pops a letter, by inductive assumption this variable occurs at

the corresponding position in sub D' and by the algorithm it pops the same letters and those letters have similar depfactors. If letters are compressed then right before the compression they have the same depfactors and in sub D' on corresponding positions there are letters with similar depfactors; in particular if one of those letters is in sub D (sub D') then both are, so the corresponding letters are compressed in the same way, also afterwards the resulting letter is still within sub D (sub D'), as it has the same depfactor as the compressed letters. The last possibility is that the depfactors get extended. If the depfactor by which this letter is extended is in sub D then the corresponding letter in sub D' gets extended by a similar depfactor and in the end those letters have similar depfactors. If this is not the case for extending of depfactors before pair compression, then the letter in question is the first (or last, which is the symmetric case) say in sub D and its depfactor gets summed with depfactor to the left. But then by the definition the first letter in sub- D' -symbols also has its depfactor summed with the depfactor to the left and so it also ceases to be a sub- D' -symbol; note that it is important here that we use endmarkers: if a letter is not an endmarker then it always has a symbol to the left and right, on the other hand the depfactor of an endmarker never gets extended. A similar argument holds for the block compression: if the sub- D -symbols have an a^ℓ prefix, so do the sub- D' -symbols. All those letters get extended with some other depfactor (which may be a different number of letters to the left) and so they all cease to be sub- D -symbols and sub- D' -symbols, respectively; a symmetric argument applies also to the b -suffix; note that here we again essentially use the ending markers in a way similar as in the pair compression. This ends the proof of the Auxiliary Claim.

Getting back to the main proof, now the (D1) follows: the D -symbols are an intersection of sup- D -symbols and sub- D -symbols; as both are intervals, also D -symbols are an interval.

Concerning (D2), consider two consecutive symbols, say a, b that have different depfactors, say D, D' , let $D = D_1 D_2 \cdots D_k$, where $D_1 \leq D_2 \leq \cdots \leq D_k$ are different basic depfactors. To show that $D \leq D'$ it is enough to show that D' is obtained by deleting a prefix of those basic depfactors and summing some suffix of next basic depfactors. So suppose for the sake of contradiction that D' includes a basic depfactor $D_0 < D_1$. As $D_0 < D_1$ the sup- D_0 -symbols are smaller than sup- D_1 -symbols, thus if a is a sup- D_1 -symbol and b a sup- D_0 -symbol then also a is a sup- D_0 -symbol, a contradiction. A symmetrical argument applies on the right end, which ends the proof. \square

3.1.2 Encoding of letters

For a depfactor D the D -letters are encoded as $D\#1, D\#2$, etc. where here D is the string in the original equation used to encode D and the numbers are given in binary; note, that there is no a priori bound on the size of such numbers. Furthermore, if $D' \sim D$ then encoding $D\#i$ and $D'\#i$ is the same (these are the same symbols by (D3)).

3.2 Pair compression strategy

As we assume that `LinWordEqSat` chooses the nondeterministic choices according to the solution, the space consumption of a particular run depends solely on the choices of the consecutive partitions chosen during pair compression. Below we describe a strategy that yields linear one.

3.2.1 Idea

Imagine we ensured that during one phase each variable popped $\mathcal{O}(1)$ letters and each basic depfactor D expanded by $\mathcal{O}(1)$ letters. Then it can be showed that $|\text{sup } D| = \mathcal{O}(1)$: on one hand we introduced $\mathcal{O}(1)$ new D -letters, say at most k , and on the other, by Lemma 2, the sup D -letters from the beginning of the phase were compressed and $1/3$ of them were removed, so there are at most $3k$ sup D letters. As a result, for each depfactor D' there are at most $3k$ letters as well (as each D' letter is a sup D -letter for some basic depfactor D). This would yield that the whole bit-space used for the encoding is linear: the numbers i used in $D'\#i$ are at most $3k = \mathcal{O}(1)$, so

they increase the size by at most a constant fraction. On the other hand, the depfactors consume:

$$\sum_{D:\text{depfactor}} \|D\| \cdot |D| = \sum_{D:\text{basic depfactor}} \|D\| \cdot |\text{sup } D|$$

(a simple proof is provided later on) and the right-hand side is linear in terms of the input equation: $|\text{sup } D| = \mathcal{O}(1)$ and $\sum_{D:\text{basic depfactor}} \|D\|$ is the bit-size of the input equation.

It remains to consider, how to ensure that a basic depfactors do not extend and variables do not pop letters. Given a phase, we call a letter *new*, if it was introduced during the pair compression. Observe that new letters cannot be popped nor can a basic depfactor be extended to a new letter. Thus they are used to prevent extending depfactors and popping: it is enough to ensure that the first/last letter of a variable is new and that a letter to the left/right of $\text{sup } D$ is new.

Unfortunately, we cannot ensure this for all variables and depfactors. What we can is to make this true *in expectation*: given a random partition there is a $1/4$ probability that a fixed pair is going to be compressed (and the resulting letter is new). This requires precise formalisation and calculations (note that the original ‘argument’ does not work immediately, when expected $\mathcal{O}(1)$ letters are popped), but indeed works.

3.2.2 Depfactors

Given a solution S of an equation we say that a variable X is *left blocked* if $S(X)$ has at most one letter or the first or second letter in $S(X)$ is new, otherwise a variable is *left unblocked*; define *right-blocked* and *right unblocked* variables similarly. A basic-depfactor D in U (or V) is *left-blocked* if in $S(U)$ (or $S(V)$, respectively) there is at most one letter to the left of $\text{sup } D$ or the letter one or two to the left of $\text{sup } D$ is new, otherwise D is *left unblocked*; define *right-blocked* and *right unblocked* basic depfactors similarly.

Lemma 8. *Consider a solution $S = S_0$ and consecutive solutions S_1, S_2, \dots corresponding to it during a phase. If a variable X becomes left (right) blocked for some S_i , then it is left (right, respectively) blocked for each S_j for $j \geq i$ and it pops to the left (right, respectively) at most 1 letter after it became left (right, respectively) blocked. If a basic depfactor D becomes left (right) blocked for some S_i then it is left (right, respectively) blocked for each S_j for $j \geq i$ and at most one letter to the left (right, respectively) will have its depfactor extended by D after D became left (right, respectively) blocked.*

Proof. If X becomes left-blocked because it has one letter, then it will stay left-blocked and can pop at most one letter further on. If it becomes left-blocked because its first or second letter is new then this new letter cannot be popped, as we pop only letters from Γ , so this letter will remain within $S(X)$ in this phase and it keep X left-blocked. In particular, if this letter is first (second) in $S(X)$, then X cannot pop left a letter (can pop at most one letter); a similar argument applies on the right-side.

Similarly, only letters from Γ (this does not include endmarkers) can have their depfactors extended, so if a letter one (or two) to the left of left-most $\text{sup } D$ -letter is new, then this depfactor can extend to the left by no or only one letter. Similarly, when there is only one letter (or no letter) to the left of $\text{sup } D$ then D can extend only by this letter and it will remain left-blocked. A symmetric argument applies for right-blocked depfactors. \square

The strategy iterates steps a, b, c, d; in a step it chooses a partition so that the corresponding

sum below decreases by 1/2, unless this sum is already 0:

$$\sum_{\substack{X \in \mathcal{X} \\ \text{left-unblocked}}} n_X \cdot \|X\| + \sum_{\substack{X \in \mathcal{X} \\ \text{right-unblocked}}} n_X \cdot \|X\| \quad (1a)$$

$$\sum_{\substack{D: \text{basic depfactor} \\ \text{left-unblocked}}} \|D\| + \sum_{\substack{D: \text{basic depfactor} \\ \text{right-unblocked}}} \|D\| \quad (1b)$$

$$\sum_{\substack{X \in \mathcal{X} \\ \text{left-unblocked}}} n_X + \sum_{\substack{X \in \mathcal{X} \\ \text{right-unblocked}}} n_X \quad (1c)$$

$$\sum_{\substack{D: \text{basic depfactor} \\ \text{left-unblocked}}} 1 + \sum_{\substack{D: \text{basic depfactor} \\ \text{right-unblocked}}} 1 \quad (1d)$$

The idea of the steps is as follows: (1a) is (roughly) the upper-bound on the increase of bit-size of depfactors in the equation after popping letters. When we decrease it, we ensure that popping increases the equation in a small way. The (1b) is a similar upper-bound on the increase due to expansion of basic depfactors. The following (1c) is connected (in a more complex way) to an increase of D -numbers after popping letters and similarly (1d) to a similar increase after the extension of depfactors.

Lemma 9. *During the pair compression LinWordEqSat can always choose a partition that at least halves the value of a chosen non-zero sum among (1a)–(1d).*

Proof. Consider (1a) and take a random partition, in the sense that each letter $a \in \Gamma$ goes to the Γ_ℓ with probability 1/2 and to Γ_r with probability 1/2. Let us fix a variable X and its side, say left. What happens with $n_X \cdot \|X\|$ in (1a) in the sum corresponding to left-unblocked variables? If X is left blocked then, by Lemma 8, it will stay left blocked and so the contribution is and will be 0. If it is left unblocked, then its two first letters a, b are not new, so they are in Γ . If $S(X)$ has only those two letters, then with probability 1/2 the a will be in Γ_r and it will be popped and X will become left-blocked (as $S(X)$ has only one letter), the same analysis applies, when the third left-most letter is new. The remaining case is that the three left-most letters in $S(X)$ are not new, let them be $a, b, c \in \Gamma$. By Lemma 3 $a \neq b \neq c$. With probability 1/4 $ab \in \Gamma_\ell \Gamma_r$ and with probability 1/4 $bc \in \Gamma_\ell \Gamma_r$. Those events are disjoint (as in one $b \in \Gamma_r$ and in the other $b \in \Gamma_\ell$) and so their union happens with probability 1/2. In both cases X will become left-blocked, as a new letter is its first or second in $S(X)$. In all uninvestigated cases the contribution of $n_X \cdot \|X\|$ cannot raise, which shows the claim in this case.

The case of (1c) is shown in the same way.

For (1b) observe that the analysis for a basic depfactor D that is left-unblocked is similar, but this time we consider the letters (in $S(U)$ or $S(V)$) to the left of $\text{sup } D$ and the depfactor D can extend to them (instead of letters being popped from variables in case of (1a)) and some of them may be compressed to one. Note that if there are no letters to the left/right then this depfactor is blocked from this side.

The case of (1d) is shown in the same way as (1b). \square

3.2.3 Space consumption

We are now ready to give the linear space bound on the size of equation, this formalises the intuition from Section 3.2.1, in particular, the argument works in the expected case. As a first step, we show w useful upper-bound on the encoding size of the equation, define

$$H_d(U, V) = \sum_{D: \text{basic depfactor}} \|D\| \cdot |\text{sup } D|$$

$$H_n(U, V) = \sum_{D: \text{basic depfactor}} 2|\text{sup } D| \cdot \log(|\text{sup } D| + 1)$$

H_d intuitively corresponds to the size of the depfactors in the encoding and H_d to the size of the D -numbers.

Lemma 10. *Given the equation (U, V) it holds that $|(U, V)| \leq H_d(U, V) + H_n(U, V)$.*

Proof. Recall that a D -letter is encoded as $D\#i$; such a number will be called D -number.

We first estimate the space used by depfactors in the encodings of all letters: Note that a D -symbol contributes $|D|$ to space usage, and when $D = D_1 D_2 \cdots D_k$, where D_1, \dots, D_k are basic depfactors, then it contributes $\sum_{i=1}^k |D_i|$; the total space usage is then obtained by taking a sum over all symbols in the equation. When we change the order of grouping and first group by a basic depfactor D' , then it is summed for $|\sup D'|$ symbols we obtain $\sum_{D': \text{basic depfactor}} |D'| \cdot |\sup D'|$. In numbers:

$$\begin{aligned} \sum_{D: \text{depfactor}} |D| \cdot |D| &= \sum_{D: \text{depfactor}} \sum_{\substack{D': \text{basic depfactor} \\ D \in \sup D'}} |D'| \cdot |D| \\ &= \sum_{D': \text{basic depfactor}} \sum_{\substack{D: \text{depfactor} \\ D \in \sup D'}} |D'| \cdot |D| \\ &= \sum_{D': \text{basic depfactor}} |D'| \cdot |\sup D'| \\ &= H_d(U, V). \end{aligned}$$

Let us now move to the space usage of D -numbers. Given a depfactor $|\sup D| = k$ each D -number is encoded on $\lceil \log(k+1) \rceil$ bits and so all D -numbers use in total $k \lceil \log(k+1) \rceil \leq 2k \log(k+1)$ bits; denote $h(x) = x \log(x+1)$. Thus, the space usage of all numbers is at most $\sum_{D: \text{depfactor}} 2h(|D|)$. Then

$$\sum_{D: \text{depfactor}} h(|D|) \leq \sum_{D: \text{basic depfactor}} h(|\sup D|). \quad (2)$$

This is easy to see: given any D -symbol in the equation, where $D = D_1 D_2 \cdots D_k$ and all D_1, D_2, \dots, D_k are basic depfactors, the D -symbol contributes $\log(|D| + 1)$ to the left-hand side of (2) and $\sum_{i=1}^k \log |D_i| + 1$ to the right-hand side. As each D -symbol is also a $\sup D_i$ symbol we have $|D| \leq |D_i|$ and thus the inequality holds. \square

Instead of showing a linear bound on $|(U, V)|$ we give a linear bound on $H(U, V)$. In the following, let (U_0, V_0) denote the input equation, note that $|U_0 V_0| \leq |(U_0, V_0)|$.

Lemma 11. *Consider an equation $U = V$ and its solution S and a run of LinWordEqSat in a phase that makes the nondeterministic choices according to S and the partitions according to the strategy, let it return an equation (U', V') . Then $H(U', V') \leq \frac{5}{6} H(U, V) + \alpha |(U_0, V_0)|$ and during the phase H on the intermediate equation is at most $\beta H(U, V) + \gamma |(U_0, V_0)|$ for some constants α, β, γ .*

Note that an estimation on α, β, γ is given explicitly in the proof.

Proof. We separately estimate the H_d and H_n .

Concerning H_d , let us first estimate the weight of basic depfactors of letters popped into the equation during a phase. For each variable we pop perhaps several letters to the left and right before block compression, but those letters are immediately replaced with single letters, so we may count it each of them as 1; also, when this side of variable becomes blocked, it can pop at most one letter. Otherwise, a side of a variable pops at most 1 letter per pair compression, in which it

is unblocked from this side. So in total the weight of popped letters is at most:

$$\underbrace{\sum_X 2n_X \cdot \|X\|}_{\text{block compression}} + \underbrace{\sum_X 2n_X \cdot \|X\|}_{\text{after } X \text{ becomes blocked}} + \sum_{I: \text{partition}} \left(\sum_{\substack{X \in \mathcal{X} \\ \text{left-unblocked in } I}} n_X \cdot \|X\| + \sum_{\substack{X \in \mathcal{X} \\ \text{right-unblocked in } I}} n_X \cdot \|X\| \right). \quad (3)$$

Observe that the third sum (the one summed over all partitions) at the beginning of the phase is equal to $\sum_X 2n_X \cdot \|X\|$, as no side of the variable is blocked, and by the strategy point (1a) its value at least halves every 4th pair compression (and it cannot increase, as by Lemma 8 no side of the variable can cease to be blocked). Thus (3) is at most

$$4 \sum_X n_X \cdot \|X\| + 8 \sum_X n_X \cdot \|X\| \left(1 + \frac{1}{2} + \frac{1}{4} + \dots \right) = 20 \sum_X n_X \cdot \|X\| \leq 20 \|(U_0, V_0)\|.$$

We now estimate, how many letters may become sup D -letters due to expansion of D , this is estimated similarly: sup D can expand to two letters during the block compression (to be more precise: perhaps many letters are popped to the left and right, but they are replaced with single letters) to one letter at each side after D becomes blocked and by one symbol for each partition in which this side of D was not blocked. So

$$\sum_{D: \text{basic depfactor}} 4\|D\| + \sum_{I: \text{partition}} \left(\sum_{\substack{D: \text{basic depfactor} \\ \text{left-unblocked in } I}} \|D\| + \sum_{\substack{D: \text{basic depfactor} \\ \text{right-unblocked in } I}} \|D\| \right) \quad (4)$$

and as in the case of (3) similarly at the beginning of the phase the second sum (so the one summed by partitions) is $\sum_{D: \text{basic depfactor}} 2\|D\| = 2\|(U_0, V_0)\|$ and it at least halves every 4th partition, by strategy point (1b). Thus similar calculations show that (4) is at most $20\|(U_0, V_0)\|$.

On the other hand, when looking at sup D for a basic depfactor D , Lemma 2 shows that the initial $|\text{sup } D|$ letters were shortened into at most $\frac{2}{3}|\text{sup } D| + 1$ till the end of the phase:

- If D corresponds to a letter, then sup D is a string of letters and Lemma 2 yields that sup D is reduced to at most $\frac{2|\text{sup } D|+1}{3}$ letters.
- If D corresponds to an ending marker, then the marker itself is unchanged and the remaining symbols in sup D are letters and Lemma 2 applies to them, so the original sup D -letters are compressed to at most $1 + \frac{2(|\text{sup } D|-1)+1}{3} < 1 + \frac{2}{3}|\text{sup } D|$.
- If D corresponds to a variable then sup D includes this variable and Lemma 2 applies to string of letter to the left and right, say of length ℓ, r , where $\ell + r = |\text{sup } D| - 1$. Then after the compression we have at most $1 + \frac{2\ell+1}{3} + \frac{2r+1}{3} = 1 + \frac{2|\text{sup } D|}{3}$ letters.

Thus:

$$\begin{aligned} H(U', V') &\leq \underbrace{40\|(U_0, V_0)\|}_{\text{new letters in depfactors}} + \underbrace{\sum_{D: \text{basic depfactor}} \|D\| \cdot \left(\frac{2}{3}|\text{sup } D| + 1 \right)}_{\text{shortened old letters}} \\ &= 40\|(U_0, V_0)\| + \sum_{D: \text{basic depfactor}} \frac{2}{3}\|D\| \cdot |\text{sup } D| + \sum_{D: \text{basic depfactor}} \|D\| \\ &= 41\|(U_0, V_0)\| + \frac{2}{3}H(U, V). \end{aligned}$$

We should also estimate, what is the maximal value of H during the phase, as somewhere in the middle we are not able to guarantee that the compression reduced the length of all letters. As we already showed that during the phase we introduce at most $40\|(U_0, V_0)\|$ bits to depfactors, this yields a bound of $H(U, V) + 40\|(U_0, V_0)\|$ bits, which shows the part of the claim of Lemma for H_d .

Concerning H_n , for a basic depfactor D and let k_D, p_D, e_D denote the numbers of: sup- D -symbols at the beginning of the phase, D -letters popped from a variable and letters to which D extended (i.e. those that become sup D -letters except popped from variables). First we estimate the sums

$$\sum_{D: \text{basic depfactor}} h(p_D) \quad \text{and} \quad \sum_{D: \text{basic depfactor}} h(e_D)$$

and then use those estimations to calculate the bound on $H_n(U', V')$. We first inspect the case of p_D ; let I_1, I_2, \dots denotes the consecutive partitions in phase. We show that

$$\sum_{D: \text{basic depfactor}} h(p_D) \leq \sum_{X \in \mathcal{X}} 25n_X + \sum_{i \geq 1} i \cdot \left(\sum_{\substack{X \in \mathcal{X} \\ \text{left-unblocked in } I_i}} n_X + \sum_{\substack{X \in \mathcal{X} \\ \text{right-unblocked in } I_i}} n_X \right). \quad (5)$$

The inequality follows as: if (one occurrence of) X popped p_X letters, then it was not blocked on left/right side for p_1/p_2 partitions, where $p_1 + p_2 \geq p_X - 4$ (note that one sequence can be popped to the left and right during block compression but it is immediately replaced with a single letter, so we treat them as one letter, also one letter can be popped to the left/right after X became blocked). Then in right-hand side of (5) the contribution from (one occurrence of) X is at least

$$25 + \frac{p_1(p_1 + 1) + p_2(p_2 + 1)}{2} \geq \frac{(p_X - 4)^2}{4} + \frac{p_X - 4}{2} + 25 \geq p_X \log(p_X + 1),$$

where the first inequality follows as $p_1 + p_2 \geq p_X - 4$ and the second can be checked by simple numerical calculation. Lastly, in (5) each p_D is equal to appropriate p_X .

The sum in braces on the right-hand side of (5) initially is at most $2\|UV\| \leq 2\|(U_0, V_0)\|$ and by strategy choice (1c) it is at least halved every 4th step. So this sum is at most:

$$\sum_{i \geq 0} \underbrace{(16i + 10)}_{4 \text{ consecutive steps}} \cdot \underbrace{2\|(U_0, V_0)\|}_{\text{initial size}} \cdot \left(\frac{1}{2}\right)^i = 104\|(U_0, V_0)\|$$

and consequently

$$\sum_{D: \text{basic depfactor}} h(p_D) \leq 129\|(U_0, V_0)\|. \quad (6)$$

The analysis for e_D is similar: when we focus on a single basic depfactor D then the estimation of amount of letters by which it extends is the same as the estimation of number of letters popped from a variable and all other calculations follow, thus we obtain that

$$\sum_{D: \text{basic depfactor}} h(e_D) \leq 129\|(U_0, V_0)\|. \quad (7)$$

Let us move to sup D letters that were present at the beginning of the phase, recall that we denote their number by k_D . Using the same analysis as in the case of bit-space used by depfactors, from Lemma 2 it follows that their number decreased by at least $\frac{k_D}{3} - 1$ in the phase due to compression. Thus

$$H_n(U', V') \leq \sum_{D: \text{basic depfactor}} h\left(\frac{2}{3}k_D + 1 + p_D + e_D\right). \quad (8)$$

Consider two subcases: first, if $\frac{2}{3}k_D + 1 + p_D + e_D \leq \frac{5}{6}k_D$, then the summand can be estimated as $h(\frac{5}{6}k_D) \leq \frac{5}{6}h(k_D)$ and we can upper bound the sum over those cases by this by $\frac{5}{6} \sum_{D: \text{basic depfactor}} h(k_D)$. On the other hand, if $\frac{2}{3}k_D + 1 + p_D + e_D > \frac{5}{6}k_D$ then $1 + p_D + e_D > \frac{1}{6}k_D$ and so $\frac{2}{3}k_D + 1 + p_D + e_D < 5(1 + p_D + e_D)$. Thus (8) is upper-bounded by:

$$H_n(U', V') \leq \frac{5}{6} \sum_{D: \text{basic depfactor}} h(k_D) + \sum_{D: \text{basic depfactor}} h(5(1 + p_D + e_D)).$$

As $h(x + y + z) \leq h(3 \max(x, y, z))$ we obtain

$$H_n(U', V') \leq \frac{5}{6} \sum_{D: \text{basic depfactor}} h(k_D) + \sum_{D: \text{basic depfactor}} h(15) + h(15p_D) + h(15e_D).$$

As $h(\alpha x) \leq (\alpha + \alpha \log(\alpha))h(x)$ for $\alpha > 1$ we are left with

$$\begin{aligned} H_n(U', V') &\leq \frac{5}{6} \sum_{D: \text{basic depfactor}} h(k_D) + \sum_{D: \text{basic depfactor}} 60 + 75h(p_D) + 75h(e_D) \\ &\leq \frac{5}{6} \sum_{D: \text{basic depfactor}} h(k_D) + 60\|(U_0, V_0)\| + 7740\|(U_0, V_0)\| + 7740\|(U_0, V_0)\| \\ &= \frac{5}{6}H_n(U, V) + 15540\|(U_0, V_0)\| \end{aligned}$$

Again, we should estimate the maximal H_n value during the phase, as inside a phase we cannot guarantee that letters get compressed, i.e. estimate $\sum_{D: \text{basic depfactor}} h(k_D + p_D + e_D)$. Using similar calculation as in the case of (8) we obtain:

$$\begin{aligned} \sum_{D: \text{basic depfactor}} h(k_D + p_D + e_D) &\leq \sum_{D: \text{basic depfactor}} h(3k_D) + h(3p_D) + h(3e_D) \\ &\leq 8 \sum_{D: \text{basic depfactor}} h(k_D) + h(p_D) + h(e_D) \\ &\leq 8H_n(U, V) + 2064\|(U_0, V_0)\| \end{aligned}$$

which shows the claim of the Lemma in the case of H_n and so also in case of H . \square

3.3 Proof of Theorem 1

We now prove Theorem 1. First of all, by Lemma 1 all our subprocedures are sound, so we never return a positive answer for an unsatisfiable equation.

Consider an equation $U = V$ at the beginning of the phase, let Γ be the set of letters in this equation. If it has a solution S' , then it also has a S over Γ such that $|S(X)| = |S'(X)|$ for each variable: we can replace letters outside Γ with a fixed letter from Γ . During the a phase we will make nondeterministic choices according to such a S .

Let S' be the corresponding solution after the phase and let the obtained equation be $U' = V'$. Then $|S'(U')| \leq \frac{2|S(U)|+1}{3}$ by Lemma 2 and we can begin the next phase with S' . Hence we will terminate after $\mathcal{O}(\log N)$ phases, where N is the length of the length-minimal solution of the input equation.

It remains to show a linear bound on the space consumption, we do that for a run that chooses the partitions according to the strategy. We show by induction that for an equation (U, V) at the beginning of a phase $H(U, V) \leq \delta\|(U_0, V_0)\|$, where $U_0 = V_0$ is the input equation and δ an appropriate constant. At the beginning $|D| = |\sup D| = 1$ for each basic depfactor, and so $H_n(D) = \|(U_0, V_0)\|$ and $H_d(U_0, V_0) = 2\|(U_0, V_0)\|$, hence the claim holds. By Lemma 11 the inequality at the end of each phase holds for $\delta = 6\alpha$ for α from Lemma 11. For intermediate

equations by the same Lemma $H(U, V)$ is at most $(6\alpha\gamma + \beta)\|(U_0, V_0)\|$, where α, β, γ are constants from the Lemma.

To bound the space size, let us also estimate other stored information: we also store the alphabet from the beginning of the phase (this is linear in the size of the equation at the beginning of the phase) and the mapping of this alphabet to the current symbols (linear in the equation at the beginning of the phase plus the size of the current equation). The terminating condition that some pair of letters in Γ^2 was not covered is guessed nondeterministically, we do not store the Γ^2 . The pair compression and block compression can be performed in linear space, see Lemma 5, this includes the change of Huffman coding.

References

- [1] Volker Diekert and Markus Lohrey. Word equations over graph products. *International Journal of Algebra and Computation*, 18(3):493–533, 2008.
- [2] Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is PSPACE-complete. *Inf. Comput.*, 202(2):105–140, 2005.
- [3] Volker Diekert, Artur Jež, and Manfred Kufleitner. Solutions of word equations over partially commutative structures. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP*, volume 55 of *LIPIcs*, pages 127:1–127:14. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2016. ISBN 978-3-95977-013-2. doi: 10.4230/LIPIcs.ICALP.2016.127. URL <http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.127>.
- [4] Volker Diekert, Artur Jež, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inf. Comput.*, 251:263–286, 2016. doi: 10.1016/j.ic.2016.09.009. URL <http://dx.doi.org/10.1016/j.ic.2016.09.009>.
- [5] Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *FOCS*, pages 112–119. IEEE Computer Society, 1998. doi: 10.1109/SFCS.1998.743434.
- [6] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.
- [7] Artur Jež. Context unification is in PSPACE. In Elias Koutsoupias, Javier Esparza, and Pierre Fraigniaud, editors, *ICALP*, volume 8573 of *LNCS*, pages 244–255. Springer, 2014. doi: 10.1007/978-3-662-43951-7_21. URL http://dx.doi.org/10.1007/978-3-662-43951-7_21.
- [8] Artur Jež. Recompression: a simple and powerful technique for word equations. *Journal of the ACM*, 63(1):4:1–4:51, Mar 2016. ISSN 0004-5411/2015. doi: 10.1145/2743014. URL <http://dx.doi.org/10.1145/2743014>.
- [9] Antoni Kościelski and Leszek Pacholski. Complexity of Makanin’s algorithm. *J. ACM*, 43(4): 670–684, 1996.
- [10] Gennadii Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).
- [11] Gennadii Makanin. Equations in a free group. *Izv. Akad. Nauk SSR, Ser. Math.* 46:1199–1273, 1983. English transl. in *Math. USSR Izv.* 21 (1983).
- [12] Yuri Matiyasevich. Some decision problems for traces. In Sergej Adian and Anil Nerode, editors, *Proceedings of the 4th International Symposium on Logical Foundations of Computer Science (LFCS’97), Yaroslavl, Russia, July 6–12, 1997*, volume 1234 of *LNCS*, pages 248–257. Springer, 1997. Invited lecture.
- [13] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725. ACM, 1999.

- [14] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi: 10.1145/990308.990312.
- [15] Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006. ISBN 1-59593-134-1.
- [16] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of word equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998. doi: 10.1007/BFb0055097.
- [17] Alexander A. Razborov. *On Systems of Equations in Free Groups*. PhD thesis, Steklov Institute of Mathematics, 1987. In Russian.
- [18] Klaus U. Schulz. Makanin’s algorithm for word equations—two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990. ISBN 3-540-55124-7. doi: 10.1007/3-540-55124-7_4.